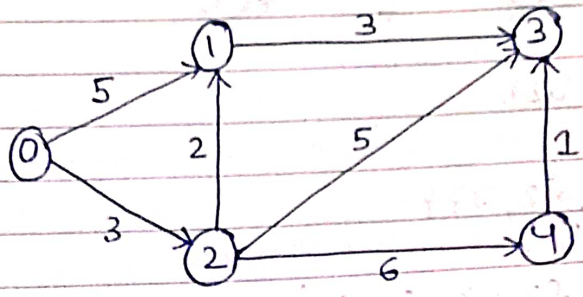
21/02/2024
Wednesday

* **Single source shortest path in directed and weighted graph –**

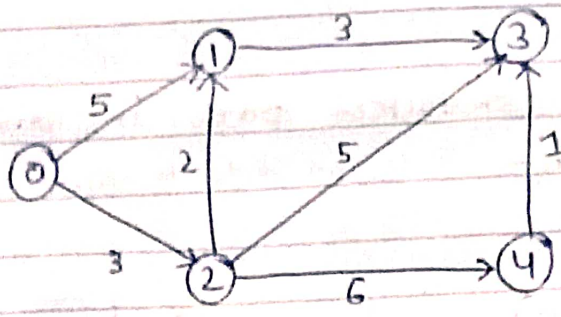


src destination	shortest path
0 → 0	0
0 → 1	5
0 → 2	3
0 → 3	8
0 → 4	9

Logic –
- Find topological ordering.
- On the basis of that topological ordering, calculate the shortest path of every node from the given source node.

Why topological ordering is needed ?

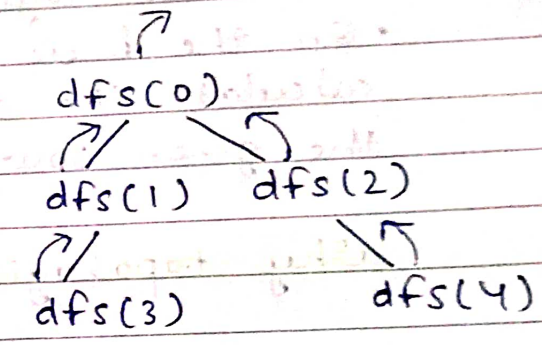
The node being processed should be independent i.e. there are no incoming edges in that node. This refers that there is no other shorter path exists for the node being worked upon. This can be achieved by using topological order of the nodes.
We can find the topological order either by dfs or bfs traversal.

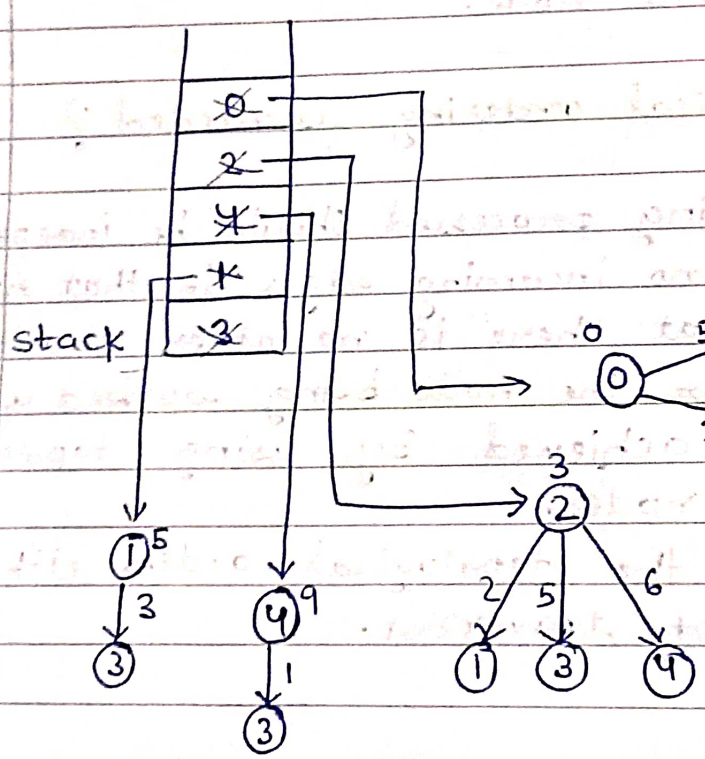


adjacency list

0 → {{1,5}, {2,3}}
1 → {{3,3}}
2 → {{1,2}, {3,5}, {4,6}}
3 → { }
4 → {{3,1}}

topological ordering





distance from src node	0	1	2	3	4
	0	5	3	8	9

Code -

```
//topological order
void topoOrderDfs (int src, unordered_map <T, bool> &
                   visited, stack <int> & st)
{
    visited[src] = true;

    for (auto nbrPair : adjList[src]){
        T nbrNode = nbrPair.first;
        int nbrDist = nbrPair.second;
        if (!visited[nbrNode]){
            topoOrderDfs(nbrNode, visited, st);
        }
    }
    st.push(src);
}
//shortest path

void shortestPath (int n, stack <T> & st){
  //distance array stores the shortest path to
  //every node from src node
    vector <int> dist (n, INT_MAX);

  //distance of src node from itself
    T src = st.top();
    dist[src] = 0;

  while (!st.empty()){

      T src = st.top();
      st.pop();
```

```
// update nbr distance for fetched node
for (auto nbrPair : adjList[src]){
    T nbrNode = nbrPair.first;
    int nbrDist = nbrPair.second;

    // update in case if shorter distance to the
    // node exists from previous one
    if (nbrDist + dist[src] < dist[nbrNode]){
        dist[nbrNode] = nbrDist + dist[src];
    }
}
}

// now distance array is ready
for(auto i : dist){
    cout << i << " ";
}
}
```

* **Dijkstra's Algorithm-**

Given is weighted graph, can be directed or undirected and a source node, find the shortest distance from source to all other nodes in graph.

It's a greedy algo. On the spot, whatever is the best, it will consider that path. Doesn't work on or approaches negative weights, negative cycles and unreachable nodes.

Algo-

(1) Make a distance vector which will store the shortest path of all nodes from the given source node.

(2) Use a priority queue or set comprises of pair where a pair's first value denotes the distance of a specific node from source node and pair's second value is the node itself.

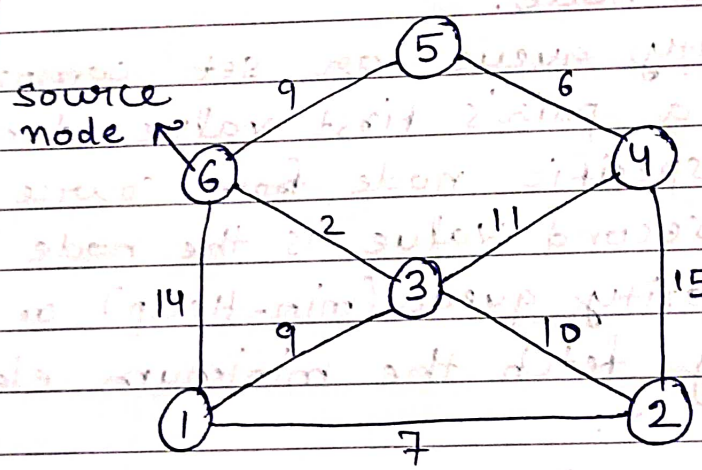
(3) Until the priority queue (min-Heap) or set is not empty, fetch the minimum element. Remove it.
Go to the adjacents or neighbours of the fetched node.
Now, check if shorter path than the previous one exists. If yes, then update the distance vector and set or priority queue accordingly

(4) After all nodes are processed i.e. set or priority queue is empty. Now, distance vector is ready which stores the shortest path of all nodes from given source node.

(Why priority queue (min heap) or set is used?

This is because they will fetch the minimum in constant (O(1)) time. As this is a greedy algo, that's why we are looking for minimum. By fetching the minimum distance everytime, we will get the shortest path from source to destination node at last.

let the graph be -



Initially, push the distance of source node from itself and source node in the set.

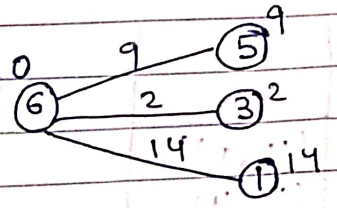
Initialise array with INT_MAX.

dist. array -

∞	∞	∞	∞	∞	∞	0
0	1	2	3	4	5	6

set -

~~(0,6)~~



pair<int, int> st

node

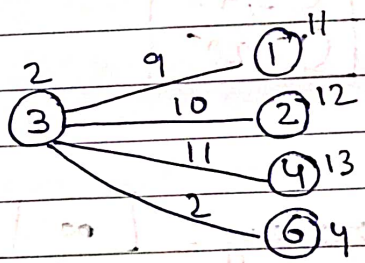
distance of node from source node

dist. array =

∞	14	∞	2	∞	9	0
0	1	2	3	4	5	6

set =

(14, 1)
~~(2, 3)~~
(9, 5)



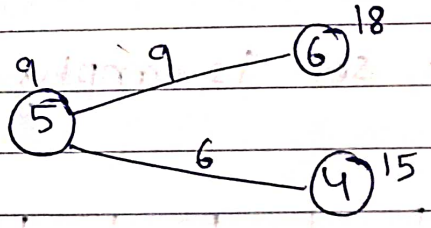
dist. array =

∞	11	12	2	13	9	0
0	1	2	3	4	5	6

set =

(11, 1)
(13, 4)
(12, 2)
~~(14, 1)~~
~~(9, 5)~~

This entry is removed because node ① ke corresponding shortest path exist krta he

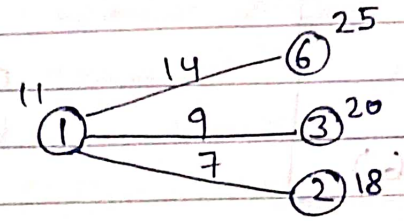


dist. array =

∞	11	12	2	13	9	0
0	1	2	3	4	5	6

set =

~~(11,1)~~
(13,4)
(12,2)

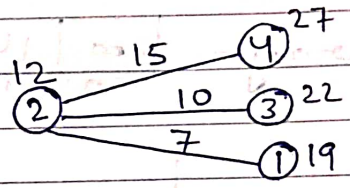


dist. array =

∞	11	12	2	13	9	0
0	1	2	3	4	5	6

set =

(13,4)
~~(12,2)~~

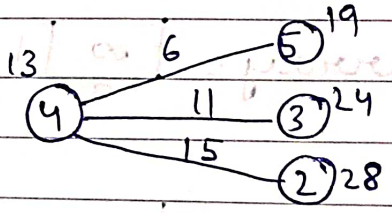


dist. array =

∞	11	12	2	13	9	0
0	1	2	3	4	5	6

set =

~~(13,4)~~



Now, set is empty and the distance array is -

dist

∞	11	12	2	13	9	0
0	1	2	3	4	5	6

Code-

```
void dijkstraAlgo (int n, T src, vector<int>& dist)
{
    set<pair<int, T>> st;

    //maintain initial state
    st.insert ({0, src});
    dist[src] = 0;

    while (!st.empty()){
        //fetch the minimum
        auto minElement = st.begin();
        pair<int, T> srcPair = *minElement;
        int srcDist = srcPair.first;
        int srcNode = srcPair.second;

        //remove the pair from set
        st.erase (minElement);

        //process the neighbour nodes
        for (auto nbrPair: adjlist[srcNode]){
            T nbrNode = nbrPair.first;
            int nbrDist = nbrPair.second;

            //if shorter distance than previous distance
            //exists, then update -
            if (srcDist + nbrDist < dist[nbrNode]){
                //fetch previous entry & remove it from set
                auto previousEntry = st.find ({dist[nbrNode]
                                               nbrNode});
                if (previousEntry != st.end()){
                    st.erase (previousEntry);
                }
```

```
        //update distance array & insert new entry
        dist[nbrNode] = srcDist + nbrDist;
        st.insert({dist[nbrNode], nbrNode});
      }
    }
  }
}
```

Time complexity - $O(E \log V)$

In the worst case, the no. of edges will be $V(V-1)$ i.e., every node is connected to every other node/vertex. So, no. of iterations in while loop will be E i.e. $V(V-1)$ in worst case. Inside loop, we are pushing, removing entries in logarithmic complexity. So, this would give $O(E \cdot \log V)$.